Multikey Quickselect*

Leonor Frias and Salvador Roura

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
lfrias@lsi.upc.edu

Abstract. In this paper we introduce Multikey Quickselect: an efficient, in-place, and easy to implement algorithm for the selection problem for strings. We present several variants of our basic algorithm, which apply to two different flavors of the selection problem. Also, we analyze the cost of the main variants, measured as the expected number of character comparisons and elements swaps. Some of the enhancements presented in this paper apply to Multikey Quicksort as well.

1 Introduction

Multikey Quicksort [1] is a well-known practical algorithm for sorting strings. Like Radixsort does, it benefits from the internal representation of strings as an array of characters to avoid redundant comparisons. But like three-way Quicksort, it partitions its input into three sets with elements less than, equal to, and greater than a given value. As a result, it makes almost as few character comparisons as Radixsort, it is easier to implement, it is in-place, and it has been shown to be rather fast in practice (see for instance, [2, 3]).

An important problem closely related to sorting is selection, i.e., finding an element of a given rank in an unsorted array. Indeed, Quickselect, the algorithm analogous to Quicksort for selection, has been thoroughly studied (for the expected number of (atomic) element comparisons and swaps see for instance [4–6]; for the expected number of character comparisons see [7]).

Quickselect is a generic algorithm that makes element comparisons as a whole, and therefore it is not very efficient for strings, in particular in the presence of long common prefixes. By contrast, Radixselect, the counterpart of Radixsort for selection, performs as few character comparisons as possible [8], but in return it requires extra linear space in the number of strings, or alternatively, two passes on the data per iteration. Linear space is needed as well to combine Quickselect with the efficient digital access techniques presented in [9–11], applied in [12], and analyzed in [13].

In this paper, we formalize Multikey Quickselect, the counterpart of Multikey Quicksort for selection. Multikey Quickselect is in-place, efficient, and easy to implement. We analyze its average number of comparisons and swaps for two flavors of the selection problem. Moreover, we propose some algorithmic enhancements that also apply to Multikey Quicksort.

* Supported by the Spanish project ALINEX (ref. TIN2005-05446)

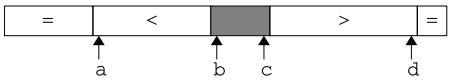

Fig. 1. 'split-end' partitioning

2 Algorithm description

We call (ternary) Multikey Quickselect the natural derivation of Multikey Quicksort for the selection problem. Algorithm 1 includes a pseudocode description. There, v is a 1-based array (or subarray) with n strings. We can consider each string as a variable-length array ending with a special value EOS. Assuming that all the strings in v are identical in the first $j-1$ characters, the algorithm returns the r-th lexicographically smallest string of v. Initially, $j = 1$.

Algorithm 1 mkqsel(v, n, j, r)

Require: $1 \leq r \leq n$
  if $n < N$ **then**
    Select and return the r-th smallest string of v using any algorithm.
  else
    Pick a partitioning value p (the so-called pivot).
    Ternary partition v on the j-th character w.r.t. p to form $v_<$, $v_=$, and $v_>$.
    (Let $n_<$, $n_=$, and $n_>$ be respectively the sizes of $v_<$, $v_=$, and $v_>$.)
    if $r \leq n_<$ **then** Return mkqsel($v_<$, $n_<$, j, r).
    else if $i > n_< + n_=$ **then** Return mkqsel($v_>$, $n_>$, j, $r - n_< - n_=$).
    else if $p \neq$ EOS **then** Return mkqsel($v_=$, $n_=$, $j+1$, $r - n_<$).
    else Return $v[r]$.
    end if
  end if

As with sorting, the pivot can be chosen in many ways. We can pick p as the j-th character from the first (say) of the remaining strings. Alternatively, the string from which to extract p could be chosen at random. The former choice is simpler, while the later is safer when the input is biased. Under our analysis hypotheses, the cost of both will turn out to be asymptotically equivalent.

Also analogously to sorting, we can use any (reasonable) selection algorithm to solve the base case when $n < N$ for some constant N.

We now adapt the 'split-end' partitioning [14] to Multikey Quickselect. The first step of this method has the invariant shown in Figure 1. Initially, $a = b = 1$ and $c = d = n$. While a string $v[b]$ with $v[b][j] > p$ is not found, increment b. Afterwards, while a string $v[c]$ with $v[c][j] < p$ is not found, decrement c. Then swap $v[b]$ with $v[c]$. Besides, swap every string x with $x[j] = p$ towards the ends, either incrementing a or decrementing d. Stop this first step when b and c cross.

At the beginning of the second step, the array consists of $v_{=1}$, $v_<$, $v_>$, and $v_{=2}$ in this order. What is due now is different from the sorting case, and depends on the variant of the selection problem that we are considering. In the 'partitioned

output' variant, the final array must become binary partitioned with respect to $v[r]$, that is, the elements in $v[1..r-1]$ must be less than or equal to $v[r]$, and the elements in $v[r+1..n]$ must be greater than or equal to $v[r]$. (This coincides with the standard definition of the selection algorithm of C++ [15].) In this variant, $v_{=1}$ must be moved after $v_<$ only when selection follows either the branch with $v_<$ or the branch with $v_=$. Symmetrically, $v_{=2}$ must be moved before $v_>$ only when selection follows either the branch with $v_>$ or the branch with $v_=$.

In the 'only selection' variant, we only ask for the value of the sought element, so less movements are required: $v_{=1}$ and $v_{=2}$ must be joined only when selection follows the branch with $v_=$. Besides, it is enough that the smaller subarray of the two is moved next to the other. No other moves are needed.

3 Analysis

In this section, we compute the cost of Multikey Quickselect. Here we assume that n infinite-long strings are drawn independently from a random uniform distribution on the universe of strings Γ^∞, where Γ is the character alphabet and $C \geq 2$ is its cardinality. Considering infinite strings is a common technical convenience; e.g. this is used in the analysis of Ternary Search Trees (TSTs) [16], whose construction is isomorphic to Multikey Quicksort.

Specifically, we calculate the asymptotic expected number of comparisons and swaps for the 'partitioned output' and the 'only selection' cases, under the usual consideration that each rank is equally likely to be selected. We assume that all swap operations have the same cost, but the analysis could be smoothly adapted to batch-swap operations [17], which roughly save a third of the moves.

3.1 Notation

Let $T_k(n)$ denote the expected cost of a call to Multikey Quickselect with n strings, where k is the number of possible values left for the current character after the previous calls. We thus have $1 \leq k \leq C$, and $k = C$ initially.

Let $t_k(n)$ be the so-called toll function, i.e., the non-recursive cost of $T_k(n)$. It includes the constant cost of picking the pivot, plus the cost of partitioning.

For every $0 \leq \ell \leq n$, let $P(n, \ell, p) = \binom{n}{\ell} p^\ell (1-p)^{n-\ell}$ be the probability that a binomial random variable with n Bernoulli trials, each with independent probability p of success, achieves exactly ℓ successes.

In the following, we use i to denote the cardinal position (starting at 0) of the value of the pivot in a subalphabet of cardinality k. For instance, if we knew from previous calls that the current n strings could only have values from the range ['e'..'h'] in the current (j-th) character, then i would range from 0 to 3.

3.2 A recurrence for the cost of ternary Multikey Quickselect

Let us consider the cost due to a recursive call into $v_<$ in Algorithm 1. Under our hypotheses, any i between 0 and $k-1$ is equally likely to be the pivot. For

every $0 \leq \ell \leq n$, the probability that exactly ℓ strings have a j-th character smaller than i is $P(n, \ell, i/k)$. Moreover, the probability that the sought string belongs to $v_<$ is ℓ/n. In that case, we must perform a recursive call with $n' = \ell$ (and implicitly $k' = i$, note that Algorithm 1 does not know nor use k).

The cost of the second recursive call in Algorithm 1 is absolutely symmetrical. Regarding the third call, the probability that exactly m strings have a j-th character equal to i is $P(n, m, 1/k)$, and the probability to follow that branch is m/n. In that case, we must perform a recursive call with $n' = m$ (and $k' = C$).

Putting all this together, and by linearity of the expectations, we get

$$T_k(n) = t_k(n) + \sum_{m=0}^{n} P\Big(n, m, \frac{1}{k}\Big) \cdot \frac{mT_C(m)}{n} + \sum_{i=1}^{k-1} \sum_{\ell=0}^{n} P\Big(n, \ell, \frac{i}{k}\Big) \cdot \frac{2\ell T_i(\ell)}{kn} \quad (1)$$

for every $n \geq N$, with $T_k(n)$ equal to some value for $1 \leq n < N$.

3.3 Toll functions

Here we list and compute the expected value of the toll functions $t_k(n)$ used in (1) under several cost measures. In the following, when we say that a string x is smaller than, equal to or greater than the pivot, we are in fact considering the result of comparing $x[j]$ against the pivot.

We start stating some useful properties of the resulting partitioning.

Lemma 1. *Consider a fixed pivot i. The expected size of $v_<$ is $n_< = in/k$. The expected size of $v_>$ is $n_> = (k - i - 1)n/k$. The expected size of $v_{=1}$ is $n_{=1} = in/(k(k-1))$ if $k > 1$, with $n_{=1} = n$ for $k = 1$. The expected size of $v_{=2}$ is $n_{=2} = (k - i - 1)n/(k(k-1))$ if $k > 1$, with $n_{=2} = 0$ for $k = 1$.*

Proof. The probabilities for a value to be equal to, smaller than or greater than the pivot are $1/k$, i/k and $(k-i-1)/k$ respectively, so we get the first two results. Moreover, we have $n_{=1} = (n_< + n_{=1})/k$ and $n_{=2} = (n_> + n_{=2})/k$. Solving the equations, we get the last two results.

Corollary 1. *Fix a pivot i. The expected size of $v_{=1}$ plus $v_<$ is $n_\leq = in/(k-1)$ if $k > 1$, with $n_\leq = n$ for $k = 1$. The expected size of $v_{=1}$ plus $v_{=2}$ is $n_= = n/k$.*

The high level description of the partitioning method uses ternary character comparisons, i.e., the outcome of a comparison is 'less than', 'equal to' or 'greater than'. The next fact holds for any string distribution and pivot picking choice.

Fact 1 *The number of ternary character comparisons is $n + o(n)$.*

In practice, ternary comparisons are usually implemented with two binary comparisons. For the sake of completeness, we count whenever a second binary comparison would be needed after a first binary comparison. This happens in the left side when a string is found to be less than or equal to the pivot, and analogously in the right side.

Lemma 2. *The expected number of 'second' binary character comparisons is* $c_k(n) = 2(k+1)n/(3k) + o(n)$ *if* $k > 1$, *with* $c_1(n) = n$.

Proof. Consider a fixed pivot i. 'Second' binary comparisons are performed in the left side with probability $(i+1)/k$. By symmetry, and using Corollary 1, $c_k(n) = \frac{1}{k}\sum_{i=0}^{k-1} 2(i+1)/k \cdot in/(k-1)$, and the lemma follows.

Let $p_k(n)$ be the expected number of swaps of the first step of the partitioning, and let $v_k(n)$ be the expected number of swaps of the second step.

Lemma 3. $p_k(n) = (k+4)n/(6k) + o(n)$ *if* $k > 1$, *with* $p_1(n) = n + o(n)$.

Proof. Fix a pivot i. Each string greater than i that before partitioning had a rank less than $n\leq$ causes one swap. Using Corollary 1, the expected number of swaps due to this source is $(k-i-1)/k \cdot in/(k-1)$. On the other hand, every string in $v_{=1}$ and $v_{=2}$ causes one swap. Averaging over the i's, the lemma follows.

Note that $v_k(n)$ depends on whether 'partitioned output' or 'only selection' is considered, while $p_k(n)$ is independent of this fact.

Lemma 4. *Assuming 'partitioned output',* $v_k(n) = 2(k+1)n/(3k^2)$ *if* $k > 1$, *with* $v_1(n) = 0$.

Proof. When $k = 1$ no swap is done. Otherwise, fix a pivot i. If the $v_<$ or the $v_=$ branch is followed, which happens with probability $(i+1)/k$, each string in $v_{=1}$ must be swapped to the middle. By symmetry with $v_{=2}$, using Lemma 1 for the expected size of $v_{=1}$, and averaging over all i's, the lemma follows.

In the following, for any boolean expression b, let $[b]$ be the Iverson bracket for b, i.e., $[b]$ evaluates to 1 when b is true and to 0 when b is false.

Lemma 5. *With 'only selection',* $v_k(n) = n/(4k^2) - n/(4k^2(k - [k \text{ is even}]))$.

Proof. When $k = 1$ no swap is done. Otherwise, swaps are performed only when the $v_=$ branch is followed, which happens with probability $1/k$. In that case, the smallest of $v_{=1}$ and $v_{=2}$ must be moved next to the other. Fix a pivot i. If k is even, we must sum (twice by symmetry) all $n_{=1}$ corresponding to $i = 0..k/2-1$. If k is odd, we must sum (again twice by symmetry) all $n_{=1}$ corresponding to $i = 0..(k-3)/2$, plus just once the $n_{=1}$ corresponding to the middle $i = (k-1)/2$. In both cases we must multiply by $1/k$, the probability of every pivot. Using the value for $n_{=1}$ in Lemma 1 and arranging things, the lemma follows.

Corollary 2. *The expected number of swaps for 'partitioned output' is* $t_k(n) = (1/6 + 4/(3k) + 2/(3k^2))n + o(n)$ *if* $k > 1$, *with* $t_1(n) = n + o(n)$.

Corollary 3. *The expected number of swaps for 'only selection' is* $t_k(n) = (1/6+2/(3k)+1/(4k^2)-1/(4k^2(k-[k \text{ is even}])))n+o(n)$, *with* $t_1(n) = n+o(n)$.

3.4 Solving the recurrence for ternary Multikey Quickselect

In this section, we solve (1) for several toll functions. We first present a technical lemma, whose proof, due to space limitations, is in the Appendix.

Lemma 6. *Suppose $t_k(n) = o(n)$ for every $1 \le k \le C$. Then $T_k(n) = o(n)$ for every $1 \le k \le C$.*

Lemma 7. *Suppose $t_k(n) = f_k \cdot n + o(n)$ for every $1 \le k \le C$. Then $T_k(n) = A_k \cdot n + o(n)$ for every $1 \le k \le C$, where A_k is defined by*

$$A_k = f_k + \frac{A_C}{k^2} + \frac{2}{k^3} \sum_{i=1}^{k-1} i^2 A_i \ . \tag{2}$$

Proof. (Sketch) Let $Z_k(n) = T_k(n) - A_k \cdot n$. Substituting into (1), we get a recurrence for the Z_k's with the same recursive calls as the recurrence for the T_k's. Using the well-known identity $\sum_{\ell=0}^{n} P(n,\ell,p)\,\ell^2 = p\,n(p\,n+1-p)$, and the definition of the A_k's, the toll function for the Z_k's turns out to be $o(n)$. Hence, by Lemma 6, $Z_k(n) = o(n)$, and the lemma follows.

Lemma 8. *Let $C \ge 2$ be any integer constant, and let f_k be any function over $[1..C]$. Then, the solution to equation (2) is*

$$A_C = \frac{(C+1)E_C}{C(C-1)} \ , \tag{3}$$

where E_k is the solution to the recurrence

$$E_k = \frac{k^3 f_k - (k-1)^3 f_{k-1}}{(k+1)k} + E_{k-1} \tag{4}$$

for $k \ge 2$, with $E_1 = f_1/2$.

Proof. (Sketch) Define $B_k = A_k - A_C/k$. This yields $B_k = f_k + \frac{2}{k^3}\sum_{i=1}^{k-1} i^2 B_i$. To solve this recurrence, define $D_k = k^3 B_k$. Then we have $D_1 = f_1$, and for $k \ge 2$, $D_k = k^3 f_k - (k-1)^3 f_{k-1} + (k+1)D_{k-1}/(k-1)$. Now let $E_k = D_k/((k+1)k)$, from which (4) is deduced. Finally, it is enough to reverse all the definitions to put the A_k's in terms of the E_k's, in particular A_C in terms of E_C.

Let $H_k = \sum_{1 \le i \le k} 1/k$ denote as usual the k-th harmonic number.

Lemma 9. *Let $C \ge 2$, and let $f_k = \alpha + \beta/k + \gamma/k^2$ for every $2 \le k \le C$, where α, β and γ are any constants. Then, the solution of (2) for A_C is*

$$A_C = 3\alpha + \frac{f_1 + 38\alpha - 10\beta + 2\gamma}{3(C-1)} + \frac{6(\beta - 3\alpha)(C+1)H_C + f_1 - \alpha - \beta - \gamma}{3C(C-1)} \ .$$

Proof. We use Lemma 8 with $f_k = \alpha + \beta/k + \gamma/k^2$ for every $k \geq 2$. Substituting into (4), we get $E_2 = (f_1 + 4\alpha + 2\beta + \gamma)/3$, and

$$E_k = 3\alpha + \frac{2\beta - 6\alpha}{k} + \frac{7\alpha - 3\beta + \gamma}{(k+1)k} + E_{k-1}$$

for $k \geq 3$. Iterating, the solution of this recurrence is

$$E_k = 3\alpha(k-2) + (2\beta - 6\alpha)\left(H_k - \frac{3}{2}\right) + (7\alpha - 3\beta + \gamma)\left(\frac{1}{3} - \frac{1}{k+1}\right) + E_2 \ .$$

Now it is enough to plug the corresponding expression for E_C into (3) and make some simplifications to finish the proof.

We need the following specific lemma to compute the expected number of swaps of 'only selection'. Its proof, omitted, is similar to that of Lemma 9.

Lemma 10. *Let $C \geq 2$, $f_1 = 0$, $f_k = 1/(k^2(k-1))$ for all even $2 \leq k \leq C$, and $f_k = 1/k^3$ for all odd $2 \leq k \leq C$. Then, the solution of (2) for A_C is*

$$A_C = \frac{4(C+1)(H_C - H_{\lfloor C/2 \rfloor}) - 2}{3C(C-1)} + \frac{(2C-1)[C \text{ is even}]}{3C(C-1)^2} - \frac{(2C+1)[C \text{ is odd}]}{3C^2(C-1)} \ .$$

Theorem 1. *The expected cost of (ternary) Multikey Quickselect is:*

- *considering (ternary) comparisons:* $(3 + \frac{13}{(C-1)} - \frac{6(C+1)H_C}{C(C-1)})n + o(n)$
- *considering 'second' binary comparisons:* $(2 + \frac{59}{9(C-1)} - \frac{24(C+1)H_C+1}{9C(C-1)})n + o(n)$
- *considering swaps (partitioned output):* $(\frac{1}{2} - \frac{14}{9(C-1)} + \frac{30(C+1)H_C-7}{18C(C-1)})n + o(n)$
- *considering swaps (only selection):*
  $(\frac{1}{2} + \frac{7}{18(C-1)} + \frac{4(C+1)H_{\lfloor C/2 \rfloor}+3}{12C(C-1)} - \frac{(2C-1)[C \text{ is even}]}{12C(C-1)^2} + \frac{(2C+1)[C \text{ is odd}]}{12C^2(C-1)})n + o(n)$

Proof. $t_k(n)$ is given in Fact 1, Lemma 2 and Corollaries 2 and 3. In all cases, $t_k(n)$ is of the form in Lemma 7. The resulting f_k are of the form in Lemmas 9 and 10. Combining everything, the theorem follows.

4 Using k in the algorithm

In the previous analyses, k was the cardinality of the remaining alphabet for the current character. In this section, we show how Multikey algorithms can benefit from the information of the value of k.

Let the global constants F and L be respectively the first and last values of the alphabet. Thus, $C = L - F + 1$. To keep track of k, we add as parameters f and l, respectively the first and last possible alphabetic values for the j-th character. This way we have $k = l - f + 1$. Initially, $f = F$ and $l = L$.

The recursive calls are modified as follows. If the $v_<$ branch is followed, then $f' = f$ and $l' = p - 1$. If the $v_>$ branch is followed, then $f' = p + 1$ and $l' = l$. If the $v_=$ branch is followed, then $f' = F$ and $l' = L$.

4.1 Specializations for small k

When $k = 1$, all the strings are equal w.r.t. the current character. Thus, we can avoid partitioning and directly proceed by the $v_=$ branch. This roughly saves n redundant ternary comparisons, and reduces in $\frac{C+1}{3C(C-1)}$ the A_C term in Lemma 7. So, as intuition suggests, this specialization is of special interest when C is small. Note that this improvement also applies to Multikey Quicksort.

When $k = 2$, one of the subarrays produced by the ternary partition will be empty. Therefore, using a binary exchange partition [18] would reduce this cost. But instead of computing the savings of this enhancement, we propose another algorithm for selecting strings, which somehow directly incorporates those specific improvements. Certainly, ternary partitioning can (and probably should) be replaced by binary partitioning when k is known.

4.2 Binary Multikey Quickselect and Quicksort

In [3], Multikey Quicksort is described as "a variation of MSD Radixsort with bucketing replaced by (three-way) Quicksort". Indeed, separating strings with equal characters guarantees algorithm progress. Nevertheless, some strings are swapped twice (with 'split-end' partitioning, those in $v_=$). Moreover, comparisons are ternary, which are in general more expensive than binary ones.

As stated above, ternary partitioning can be replaced by binary partitioning if k is know. We call the resulting algorithms *binary Multikey Quickselect* (see Algorithm 2) and *binary Multikey Quicksort*. Algorithm progress is guaranteed by choosing a pivot $p > b$, and incrementing j when $k = 1$.

A possible pivot selection algorithm is: scan the array from left to right, until a value greater than f is found. If it is not found, proceed as when $k = 1$. Otherwise, partition the part of the array not yet examined, as the strings that have been discarded as pivots are already in a correct place. Note that binary Multikey Quickselect as described is directly a 'partitioned output' variant.

Algorithm 2 mkqsel_binary(v, n, j, r, f, l)

Require: $1 \leq r \leq n$
  if $n < N$ **then**
    Select and return the r-th smallest string of v using any algorithm.
  else if $l - f \geq 0$ and exists a partitioning value $p > f$ **then**
    Binary partition v on the j-th character w.r.t. p to form $v_<$ and $v_\geq$.
    (Let $n_<$ and $n_\geq$ be respectively the sizes of $v_<$ and $v_\geq$.)
    if $i \leq n_<$ **then** Return mkqsel_binary($v_<$, $n_<$, j, r, f, $p - 1$).
    else Return mkqsel_binary($v_\geq$, $n_\geq$, j, $r - n_<$, p, l).
  else if $b \neq$ EOS **then** Return mkqsel_binary(v, n, $j + 1$, r, F, L).
  else Return $v[r]$.
  end if

5 Analysis of binary Multikey Quickselect

In this section, we analyze binary Multikey Quickselect using steps very similar to those in Section 3. Therefore, we only point out the main differences.

Let $X_k(n)$ denote the expected cost of a call to binary Multikey Quickselect with n elements, where k is the current number of possible characters. To start with, $X_1(n) = X_C(n)$. For $k \geq 2$, any i between 1 and $k-1$ is equally likely to be the pivot. A call into $v_<$ will have $k' = i$, and a call into $v_\geq$ will have $k' = k - i$. Taking into account symmetry, this time we get the recurrence

$$X_k(n) = x_k(n) + \sum_{m=0}^{n} P\Big(n, m, \frac{1}{k}\Big) \cdot \frac{2mX_C(m)}{(k-1)n} + \sum_{i=2}^{k-1} \sum_{\ell=0}^{n} P\Big(n, \ell, \frac{i}{k}\Big) \cdot \frac{2\ell X_i(\ell)}{(k-1)n} \quad (5)$$

for $2 \leq k \leq C$ and every $n \geq N$, with $X_k(n)$ equal to some value for $1 \leq n < N$.

5.1 Toll functions

Here we compute the expected value of the toll functions $x_k(n)$ used in (5). Note that they hold for $k > 1$ (for $k = 1$ they are 0). As for ternary partitioning, the following fact holds for any string distribution and pivot selection algorithm.

Fact 2 *The number of (binary) character comparisons is $n + o(n)$.*

Lemma 11. *The expected number of swaps is $x_k(n) = (k+1)n/(6k) + o(n)$.*

Proof. Consider a fixed pivot $i > 0$. Each element that before partitioning had a rank less than $n_<$ and it is greater or equal than i causes one swap. The probability of this event is $\frac{k-i}{k}$ for $n_<$ elements. Besides, the probability that an element is smaller than i is $\frac{i}{k}$ and thus, $n_< = (in)/k$. As a result, the expected number of swaps is $(k-i)(in)/k^2$. Averaging over all i's the lemma follows.

5.2 Solving the recurrence for binary Multikey Quickselect

Here we solve (5) for several toll functions. All proofs are omitted, because they are similar (and somehow simpler) than those in Section 3.

Lemma 12. *Suppose $x_k(n) = o(n)$ for every $1 \leq k \leq C$. Then $X_k(n) = o(n)$ for every $1 \leq k \leq C$.*

Lemma 13. *Suppose $x_k(n) = f_k \cdot n + o(n)$ for every $1 \leq k \leq C$. Then $X_k(n) = A_k \cdot n + o(n)$ for every $1 \leq k \leq C$, where A_k is defined by*

$$A_k = f_k + \frac{2A_C}{k^2(k-1)} + \frac{2}{k^2(k-1)} \sum_{i=2}^{k-1} i^2 A_i \quad (6)$$

for $k \geq 2$, with $A_1 = A_C$.

Lemma 14. *Let $C \geq 2$ be any integer constant, and let f_k be any function over $[2..C]$. Then, the solution to (6) is $A_C = E_C/(C-1)$, where E_k is the solution to the recurrence*

$$E_k = kf_k - \frac{(k-1)(k-2)f_{k-1}}{k} + E_{k-1} \tag{7}$$

for $k \geq 3$, with $E_2 = 2f_2$.

Lemma 15. *Let $C \geq 2$, and let $f_k = \alpha + \beta/k + \gamma/k^2$ for every $2 \leq k \leq C$, where α, β and γ are any constants. Then, the solution of (6) for A_C is*

$$A_C = 3\alpha + \frac{2(\beta-\alpha)(H_C-1)}{C-1} + \frac{\gamma}{C}\,.$$

Theorem 2. *On the average, (binary) Multikey Quickselect performs $\frac{1}{2}n + o(n)$ swaps and $(3 - \frac{2(H_C-1)}{C-1})n + o(n)$ comparisons.*

6 Algorithm comparison and implementation issues

The results of our analyses allow us to quantitatively compare ternary and binary Multikey Quickselect between them, and also against other algorithms. Figure 2 summarizes in graphical form the results presented for Multikey Quickselect in this paper. It also includes the results for (binary) Quickselect for uniformly random inputs. In particular, we plot the number of character comparisons using [7, Theorem 2 and Figure 1].

The average number of ternary comparisons in Ternary Multikey Quickselect and of binary comparisons in Binary Multikey Quickselect is never greater than $3n$ and tends to this value when C goes to infinity. Indeed, this coincides with the average number of key comparisons in Quickselect (see for instance [5]). Note that the number of character comparisons of our algorithms is smaller than the number of *key* comparisons of Quickselect. For ternary Multikey Quickselect, this fact holds only if ternary comparisons are regarded as atomic. The average number of character comparisons in Quickselect is a decreasing function that tends to $3n$ when C goes to infinity, as expected. For small alphabets, common prefixes are relatively frequent, so Quickselect does not behave very well.

With respect to swaps, Binary Multikey Quicksort is clearly advantageous. This is especially the case for small alphabet cardinalities and when a partitioned output is required. The average number of swaps tends to $n/2$ when C goes to infinity. Analogously to comparisons, this coincides with the average number of swaps in Quickselect (see for instance [6]).

We have made the analysis for random inputs because some tractable model must be fixed. But arguably, other input sources should be considered to better compare among selection algorithms for strings. However, note that a random source is likely to be the less appropriate for our algorithms, which are designed to adapt well to the existence of long common prefixes of many real datasets.

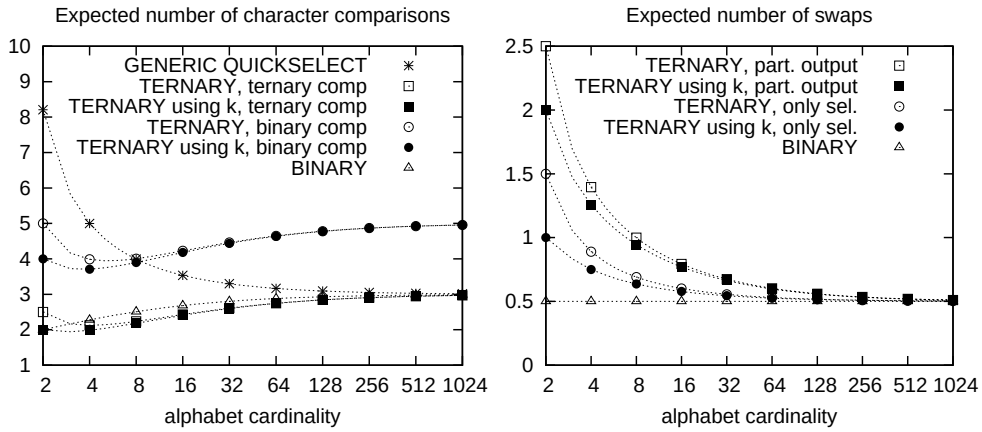

Fig. 2. Summary of the analysis results for Multikey Quickselect

Some of our algorithms need the alphabet cardinality C to compute k. If C is not provided, an upper bound can be computed from the value data type. Note that, in practice, not all the k possible values for the next character may be present. However, as actual pivots are got from the input, at most one step is required to get exact bounds and algorithm progress is guaranteed. The cost in this case would be slightly (unless C is small) higher than the analyzed in this paper.

Alternatively, we could choose as pivot a character chosen uniformly at random in the current range. Under our source model, this strategy would have the same cost as our algorithms, but would not be robust for general inputs.

A C++ implementation of our algorithms is available under `www.lsi.upc.edu/~lfrias/research/mkqsel.zip` . It follows from the Multikey Quicksort in [1].

7 Conclusions and further work

We have presented Multikey Quickselect, a practical selection algorithm for strings, which combines radix access with Quickselect-like partitioning. Multikey Quickselect had not been formalized before, and its properties cannot be directly deduced from previous analysis. We have described several variants of the algorithm, including ternary and binary partitioning, and we have provided a detailed analysis for the expected number of comparisons and swaps. We also have shown how to specialize the basic algorithm to avoid useless operations.

Multikey Quicksort could also take advantage of these ideas. In particular, the expected number of swaps when sorting can be computed by techniques similar to those used in this paper. Other approaches include considering alternative key distributions and pivot selection methods.

Finally, our implementation could be the starting point of an experimental analysis on Multikey algorithms. To the best of our knowledge, experimental results on string selection are scarce, and non-existent for Radixselect.

References

1. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1997) 360–369
2. Sinha, R., Wirth, A.: Engineering burstsort: Towards fast in-place string sorting. In McGeoch, C.C., ed.: WEA. Volume 5038 of Lecture Notes in Computer Science., Springer (2008) 14–27
3. Andersson, A., Nilsson, S.: Implementing radixsort. Journal on Experimental Algorithmics **3**(7) (1998)
4. Knuth, D.E.: The art of computer programming, volume 3 (3rd ed.): Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1997)
5. Mahmoud, H.M., Modarres, R., Smythe, R.T.: Analysis of quickselect: An algorithm for order statistics. ITA **29**(4) (1995) 255–276
6. Martínez, C., Roura, S.: Optimal sampling strategies in quicksort and quickselect. SIAM Journal on Computing **31**(3) (2001) 683–705
7. Valle, B., Clment, J., Fill, J.A., Flajolet, P.: The number of symbol comparisons in quicksort and quickselect. In: 36th International Colloquium on Automata, Languages and Programming (ICALP 2009). Volume 5555 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer-Verlag (2009) 750–763
8. Mahmoud, H., Flajolet, P., Jacquet, P., Régnier, M.: Analytic variations on bucket selection and sorting. Acta Informatica **36**(9-10) (2000) 735–760
9. Austern, M.H., Stroustrup, B., Thorup, M., Wilkinson, J.: Untangling the balancing and searching of balanced binary search trees. Software: Practice and Experience **33**(13) (2003) 1273–1298
10. Roura, S.: Digital access to comparison-based tree data structures and algorithms. Journal of Algorithms **40**(1) (2001) 1–23
11. Grossi, R., Italiano, G.F.: Efficient techniques for maintaining multidimensional keys in linked data structures. In: ICALP '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming, London, UK, Springer-Verlag (1999) 372–381
12. Frias, L., Petit, J.: Combining digital access and parallel partition for quicksort and quickselect. In: IWMSE '09: Proceedings of the 2nd international workshop on Multicore software engineering, New York, NY, USA, ACM (2009) To appear.
13. Frias, L.: On the number of string lookups in BSTs (and related algorithms) with digital access. Technical report LSI-09-14-R, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics (2009)
14. Bentley, J.L., McIlroy, M.D.: Engineering a sort function. Software: Practice and Experience **23**(11) (1993) 1249–1265
15. International Standard ISO/IEC 14882: Programming languages — C++. 1st edn. American National Standard Institute (1998)
16. Clément, J., Flajolet, P., Vallée, B.: Dynamical sources in information theory: A general analysis of trie structures. Algorithmica **29**(1) (2001) 307–369

17. Kim, K.P.E.: Improving multikey quicksort for sorting strings with many equal elements. Information Processing Letters (January 2009)
18. Mcilroy, P.M., Bostic, K., Mcilroy, M.D.: Engineering radix sort. Computing Systems **6** (1993) 5–27

Appendix (proofs for the referees)

Assume $T_0(n) = 0$ to simplify some expressions below. The expected contribution of the calls to $v_<$ when the character i is chosen as pivot is

$$S_1\{T\}(i,k,n) = \sum_{\ell=0}^{n} P(n,\ell,i/k) \cdot \frac{\ell}{n} \cdot T_i(\ell) \ .$$

(Note that we use the notation shown above to compact some proofs in this appendix. For instance, $S_1\{Y\}(i,k,n)$ would denote the same expression replacing each $T_i(\ell)$ by $Y_i(\ell)$, and $S_1\{|Z|\}(i,k,n)$ would denote the same expression replacing each $T_i(\ell)$ by $|Z_i(\ell)|$.) Similarly, for the calls to $v_>$ we have

$$S_2\{T\}(i,k,n) = \sum_{r=0}^{n} P(n,r,(k-i-1)/k) \cdot \frac{r}{n} \cdot T_{k-i-1}(r) \ ,$$

and for the calls to $v_=$ (which do not depend on i) we have

$$S_3\{T\}(k,n) = \sum_{m=0}^{n} P(n,m,1/k) \cdot \frac{m}{n} \cdot T_C(m) \ .$$

Altogether, by linearity of expectations,

$$T_k(n) = t_k(n) \ + \ \frac{1}{k} \sum_{i=0}^{k-1} \Big(S_1\{T\}(i,k,n) \ + \ S_2\{T\}(i,k,n) \ + \ S_3\{T\}(k,n) \Big) \ .$$

Let

$$S\{T\}(k,n) = S_3\{T\}(k,n) \ + \ \frac{2}{k} \sum_{i=1}^{k-1} S_1\{T\}(i,k,n) \ .$$

Then, by symmetry, we get

$$T_k(n) = t_k(n) \ + \ S\{T\}(k,n)$$

for every $n \geq N$, with $T_k(n)$ equal to any value for $1 \leq n < N$.

This lemma could be considered as a part of Lemma 6.

Lemma 16. *Suppose $t_k(n) = O(n)$ for every $1 \le k \le C$. Then $T_k(n) = O(n)$ for every $1 \le k \le C$.*

Proof. Using the hypothesis, for every $1 \le k \le C$ there exist $n_k \ge N$ and B_k large enough such that $|t_k(n)| \le B_k n$ for every $n \ge n_k$. Let $N' = \max\{2(C+1), n_1, n_2, \ldots, n_C\}$, let $B' = \max\{2k^2 B_k/(C-1) : 1 \le k \le C\}$, let $B'' = \max\{|kT_k(n)|/(Cn) : 1 \le k \le C, 1 \le n < N'\}$, and let $B = \max\{B', B''\}$. With all these definitions, what we get is that for all $1 \le k \le C$, if $n \ge N'$ then $|t_k(n)| \le B(C-1)n/(2k^2)$; and if $n < N'$ then $|T_k(n)| \le BCn/k$. These properties will be useful in some steps below.

Now let $Y_k(n) = |T_k(n)| - BCn/k$ for all $1 \le k \le C$. Substituting into (1) and using the definitions of $S\{T\}(k,n)$, $S_3\{T\}(k,n)$ and $S_1\{T\}(i,k,n)$, for $n \ge N'$ we have $Y_k(n) \le I_k(n) + S\{Y\}(k,n)$, where

$$I_k(n) = B(C-1)n/(2k^2) \;+\; \sum_{m=0}^{n} P(n,m,1/k) \cdot \frac{m}{n} \cdot BCm/C$$
$$+\; \frac{2}{k} \sum_{i=1}^{k-1} \sum_{\ell=0}^{n} P(n,\ell,i/k) \cdot \frac{\ell}{n} \cdot BC\ell/i \;-\; BCn/k\,.$$

Therefore,

$$2k^2 I_k(n)/B = (C-1)n \;+\; \frac{2k^2}{n} \sum_{m=0}^{n} P(n,m,1/k) \cdot m^2$$
$$+\; \frac{4Ck}{n} \sum_{i=1}^{k-1} \frac{1}{i} \sum_{\ell=0}^{n} P(n,\ell,i/k) \cdot \ell^2 \;-\; 2Ckn\,.$$

For the next step, we use the well-known identity

$$\sum_{c=0}^{n} P(n,c,p)\, c^2 = p\, n(p\, n + 1 - p) \tag{8}$$

for any $0 \le p \le 1$ to get

$$\sum_{m=0}^{n} P(n,m,1/k) \cdot m^2 = \frac{n(n+k-1)}{k^2}\,,$$

and also

$$\sum_{i=1}^{k-1} \frac{1}{i} \sum_{\ell=0}^{n} P(n,\ell,i/k) \cdot \ell^2 \;=\; \sum_{i=1}^{k-1} \frac{n(in+k-i)}{k^2} \;=\; \frac{(k-1)(n+1)n}{2k}\,.$$

Putting all this together produces

$$2k^2 I_k(n)/B \;=\; (1-C)n + 2(C+1)(k-1) \;\le\; (C-1)(2(C+1)-n) \;\le\; 0\,,$$

which implies $I_k(n) \leq 0$ and $Y_k(n) \leq S\{Y\}(k,n)$ for every $n \geq N'$. Furthermore, we have $Y_k(n) \leq 0$ for every $n < N'$. Hence, a trivial proof by induction on n shows that $Y_k(n) \leq 0$ for every n, from which we deduce $|T_k(n)| \leq BCn/k = O(n)$ for every n and every k.

Proof. **(Lemma 6)**

For a binomial random variable $R(n,p)$ with n trials and probability p, and for every $\varepsilon > 0$, a Chernoff bound yields

$$\Pr\{R(n,p) \geq (p+\varepsilon)n\} \leq e^{-\varepsilon^2 n/2} \; .$$

For every $1 \leq k \leq C$ and every $1 \leq i < k$, the probabilities in the expression for $S_1\{T\}(i,k,n)$ are those for $R(n,i/k)$. Let $f = 1-1/(2C)$. Substituting above with $\varepsilon = f - p$, we get the bound $\Pr\{R(n,p) \geq fn\} \leq e^{-(f-p)^2 n/2} \leq e^{-n/(8C^2)}$.

On the other hand, for every i we can split the recurrence for $S_1\{|T|\}(i,k,n)$ into two sets of recursive calls, like this:

$$\begin{aligned} S_1\{|T|\}(i,k,n) = & \sum_{0 \leq \ell \leq fn} P(n,\ell,i/k) \cdot \frac{\ell}{n} \cdot |T_i(\ell)| \\ & + \sum_{fn < \ell \leq n} P(n,\ell,i/k) \cdot \frac{\ell}{n} \cdot |T_i(\ell)| \; . \end{aligned}$$

Taking into account $\ell \leq n$, and $|T_i(\ell)| = O(n)$ by Lemma 16, the second sum above can be bounded by $e^{-n/(8C^2)}O(n) = o(n)$. The same bound can be proved for the recursive calls to $|T_C(m)|$ with $m \geq fn$ in the expression for $S_3\{|T|\}(k,n)$.

Now define $M(n) = \max\{|T_k(n)| : 1 \leq k \leq C\}$. Let a_n be any index between 1 and fn where $M(a_n)$ is maximum. Then

$$\begin{aligned} M(n) &\leq \max\{\ |t_k(n)| + S\{|T|\}(k,n) : 1 \leq k \leq C\ \} \\ &\leq o(n) + M(a_n) \max_{1 \leq k \leq C} \left\{ \sum_{0 \leq m < fn} P(n,m,1/k) \cdot \frac{m}{n} \right. \\ &\qquad \left. + \; \frac{2}{k} \sum_{i=1}^{k-1} \sum_{0 \leq \ell < fn} P(n,\ell,i/k) \cdot \frac{\ell}{n} \right\} , \end{aligned}$$

because $|T_C(m)| \leq M(m) \leq M(a_n)$ and also $|T_i(\ell)| \leq M(\ell) \leq M(a_n)$. Taking into account that the sum of weights for each k is bounded by 1, we get $M(n) \leq o(n) + M(a_n)$. Now, the solution to the recurrence $M(n) = o(n) + M(a_n)$ is $M(n) = o(n)$ —see Theorem 5.3 and Lemma 5.4 of [Roura, S. 2001. Improved master theorems for divide-and-conquer recurrences. J. ACM 48, 2 (Mar. 2001), 170-205.], for instance. Since $|T_k(n)| \leq M(n)$, this finishes the proof.

Proof. **(Lemma 7)**

Let $Z_k(n) = T_k(n) - A_k \cdot n$ for all $1 \le k \le C$. Substituting into (1) and using the definitions of $S\{T\}(k,n)$, $S_3\{T\}(k,n)$ and $S_1\{T\}(i,k,n)$, for $n \ge N'$ we have $Z_k(n) = J_k(n) + S\{Z\}(k,n)$, where

$$J_k(n) = f_k \cdot n + o(n) \;+\; \sum_{m=0}^{n} P(n,m,1/k) \cdot \frac{m}{n} \cdot A_C \cdot m$$
$$+\; \frac{2}{k} \sum_{i=1}^{k-1} \sum_{\ell=0}^{n} P(n,\ell,i/k) \cdot \frac{\ell}{n} \cdot A_i \cdot \ell \;-\; A_k \cdot n\,.$$

Using (8), the first sum above is equal to $A_C/k^2(n+k-1)$, and the second sum above is equal to $2/k^3 \sum_{i=1}^{k-1} i^2 A_i(n+k/i-1)$. Therefore, by the definition of the A_k's in (2),

$$J_k(n) \;=\; o(n) + \frac{A_C(k-1)}{k^2} + \frac{2}{k^3} \sum_{i=1}^{k-1} i A_i(k-i) \;=\; o(n)\,.$$

Thus we have $Z_k(n) = o(n) + S\{Z\}(k,n)$, which is $o(n)$ by Lemma 6. The lemma follows.

Lemma 17 (Lemma 8 extended). *Let $C \geq 2$ be any integer constant, and let f_k be any function over $[1..C]$. Then, the solution to (2) is*

$$A_k = \frac{(k+1)E_k}{k^2} + \frac{(C+1)E_C}{C(C-1)k} \,, \tag{9}$$

where E_k is the solution to the recurrence

$$E_k = \frac{k^3 f_k - (k-1)^3 f_{k-1}}{(k+1)k} + E_{k-1}$$

for $k \geq 2$, with $E_1 = f_1/2$. In particular,

$$A_C = \frac{(C+1)E_C}{C(C-1)} \,.$$

Proof. First, note that every A_k depends on A_C. To remove this dependence, define $B_k = A_k - A_C/k$, which yields

$$\begin{aligned} B_k &= f_k + \frac{A_C}{k^2} + \frac{2}{k^3}\sum_{i=1}^{k-1} i^2 \left(B_i + \frac{A_C}{i}\right) - \frac{A_C}{k} \\ &= f_k + \frac{2}{k^3}\sum_{i=1}^{k-1} i^2 B_i + \left(1 - k + \frac{2}{k}\sum_{i=1}^{k-1} i\right)\frac{A_C}{k^2} \\ &= f_k + \frac{2}{k^3}\sum_{i=1}^{k-1} i^2 B_i \,. \end{aligned}$$

This is a recurrence that can be solved by more or less standard manipulations. To begin with, define

$$D_k \;=\; k^3 B_k \;=\; k^3 f_k + 2\sum_{i=1}^{k-1} \frac{D_i}{i} \,.$$

Then we have $D_1 = f_1$, and for every $k \geq 2$,

$$D_k - D_{k-1} = k^3 f_k - (k-1)^3 f_{k-1} + \frac{2D_{k-1}}{k-1} \,,$$

that is,

$$D_k = k^3 f_k - (k-1)^3 f_{k-1} + \frac{(k+1)D_{k-1}}{k-1} \,.$$

As a last step, let

$$E_k \;=\; \frac{D_k}{(k+1)k} \;=\; \frac{k^3 f_k - (k-1)^3 f_{k-1}}{(k+1)k} + E_{k-1}$$

for $k \geq 2$, with $E_1 = f_1/2$. Now we observe that $D_k = (k+1)kE_k$, $B_k = (k+1)E_k/k^2$, and $A_k = (k+1)E_k/k^2 + A_C/k$. Hence, we can deduce $A_C = (C+1)E_C/C^2 + A_C/C$, which yields $A_C = (C+1)E_C/(C(C-1))$.

Lemma 18 (Lemma 9 extended). *Let $C \geq 2$, and let $f_k = \alpha + \beta/k + \gamma/k^2$ for every $2 \leq k \leq C$, where α, β and γ are any constants. Then, the solution to (2) is*

$$A_C = 3\alpha + \frac{f_1 + 38\alpha - 10\beta + 2\gamma}{3(C-1)} + \frac{6(\beta - 3\alpha)(C+1)H_C + f_1 - \alpha - \beta - \gamma}{3C(C-1)} ,$$

$A_1 = f_1 + A_C$, *and*

$$A_k = 3\alpha + \frac{3A_C + f_1 + 29\alpha - 10\beta + 2\gamma}{3k} + \frac{6(\beta - 3\alpha)(k+1)H_k + f_1 - \alpha - \beta - \gamma}{3k^2}$$

for $2 \leq k < C$. (This last expression also holds for $k = C$.)

Proof. We use Lemma 8 with $f_k = \alpha + \beta/k + \gamma/k^2$ for every $k \geq 2$. Substituting into (4), we get $E_2 = (f_1 + 4\alpha + 2\beta + \gamma)/3$, and

$$\begin{aligned} E_k &= \frac{\alpha(3k^2 - 3k + 1) + \beta(2k-1) + \gamma}{(k+1)k} + E_{k-1} \\ &= 3\alpha + \frac{2\beta - 6\alpha}{k} + \frac{7\alpha - 3\beta + \gamma}{(k+1)k} + E_{k-1} \end{aligned}$$

for $k \geq 3$. Iterating, the solution of this recurrence is

$$E_k = 3\alpha(k-2) + (2\beta - 6\alpha)\left(H_k - \frac{3}{2}\right) + (7\alpha - 3\beta + \gamma)\left(\frac{1}{3} - \frac{1}{k+1}\right) + E_2 .$$

Note that this last equality holds for $k \geq 2$. Now it is enough to plug this expression and the corresponding one for E_C into (9) and (3) and make some simplifications to finish the proof.

Proof. **(Lemma 10)** Again, we use Lemma 8, this time with $f_1 = 0$, $f_k = 1/(k^2(k-1))$ for even k, and $f_k = 1/k^3$ for odd $k > 1$. Substituting into (4), we get $E_1 = 0$, $E_2 = 1/3$, and

$$E_k = \frac{k/(k-1) - 1}{(k+1)k} + \frac{1 - (k-2)/(k-3)}{k(k-1)} + E_{k-2}$$

for even $k > 2$. From this, and after a lengthy manipulation, we can get

$$E_k = \frac{4(H_k - H_{k/2}) - 2}{3} + \frac{2k-1}{3(k+1)(k-1)}$$

for even $k > 2$. (Alternatively, this can be proved by induction.) Now, for odd $k > 1$, it is enough to use

$$E_k = \frac{1 - (k-1)/(k-2)}{(k+1)k} + E_{k-1}$$

and simplify the resulting expression to get

$$E_k = \frac{4(H_k - H_{\lfloor k/2 \rfloor}) - 2}{3} - \frac{2k+1}{3(k+1)k} \, .$$

From here, it is easy to obtain the result claimed for A_C.

As stated in the paper, the proofs of the lemmas for the cost of binary Multikey Quickselect are very similar, in fact a bit easier, than the corresponding proofs for the cost of ternary Multikey Quickselect. Therefore, we only give (or sketch) the couple of proofs that have some interest.

Proof. **(Lemma 14)**

We follow exactly the same steps as in Lemma 8, this time with $B_k = A_k - A_C/k$, which yields $B_k = f_k + \frac{2}{k^2(k-1)} \sum_{i=2}^{k-1} i^2 B_i$, $D_k = k^2(k-1)B_k$, which yields $D_k = k^2(k-1)f_k - (k-1)^2(k-2)f_{k-1} + kD_{k-1}/(k-2)$, and $E_k = D_k/(k(k-1))$, which yields (7).

Lemma 19 (Lemma 15 extended). *Let $C \geq 2$, and let $f_k = \alpha + \beta/k + \gamma/k^2$ for every $2 \leq k \leq C$, where α, β and γ are any constants. Then, the solution of (6) is*

$$A_C = 3\alpha + \frac{2(\beta - \alpha)(H_C - 1)}{C - 1} + \frac{\gamma}{C} ,$$

and

$$A_k = 3\alpha + \frac{2(\beta - \alpha)H_k - \alpha - 2\beta}{k} + \frac{\gamma(k-1)}{k^2} + \frac{A_C}{k}$$

for $2 \leq k < C$. (This last expression also holds for $k = C$.)

Proof. We use Lemma 14 with $f_k = \alpha + \beta/k + \gamma/k^2$ for every $k \geq 2$. Substituting into (7), we get $E_2 = 2\alpha + \beta + \gamma/2$, and

$$E_k = 3\alpha + \frac{2\beta - 2\alpha}{k} + \frac{\gamma}{k(k-1)} + E_{k-1}$$

for $k \geq 3$. Iterating, the solution of this recurrence is

$$E_k = 3\alpha(k-2) + (2\beta - 2\alpha)\left(H_k - \frac{3}{2}\right) + \gamma\left(\frac{1}{2} - \frac{1}{k}\right) + E_2 .$$

From this, it is easy to obtain the values for A_k.